Distributed Processing in PGMT

by

Richard S. Stevens

August 13, 2002

1. Abstract

At the Naval Research Laboratory (NRL) we designed and implemented a system to support the execution of dataflow graphs on a network of processors. We use a scheme of automatic static assignment of nodes to processors. During execution, the system monitors performance. If necessary, a different static assignment is chosen to improve performance. Reassignment occurs "on the fly" while execution continues with little interruption. The continual monitoring of performance and reassignment continues until an assignment with acceptable performance is achieved. In this paper we describe how we accomplish the change from one static reassignment to another with little interruption of execution.

2. Introduction

At NRL (Naval Research Laboratory) we wrote the PGM (Processing Graph Method) Specification [1] to describe the basic features of a support system for developing applications for a distributed network of processors. In the PGMT (PGM Tool) project we developed an implementation of PGM. This project was conducted with the support of the Advanced Systems Technology Office for Undersea Warfare in the Office of Naval Research.

To understand this document, we recommend that the reader be familiar with the PGM Specification. However we assume only that the reader is familiar with the concept of dataflow [2].

The motivation behind dataflow is to provide a model that permits the representation of an application with parallel processing in order to support the use of a network of processors for concurrent processing and hence increased throughput. The basis of dataflow is a directed bipartite graph. As in the theory of Petri Nets [3], we call the two sets of nodes *transitions* and *places*. In dataflow, the transitions are typically called *nodes*, and the places, represented by directed arcs, are typically called *queues*. Henceforth, unless otherwise stated, we will refer to them as *transitions* and *places*, respectively.

In the development of PGMT, one of the most interesting and challenging problems we encountered was the design of distributed processing. We chose a scheme that we call *quasi-static assignment*.

In static assignment, each transition is permanently assigned to a specific processor in the network. Every time a specific transition executes, it does so in its assigned processor

Suppose a transition A executes in its assigned processor P_A and produces data to a place Q for execution by another transition B, assigned to processor P_B. Then the data that A produces may be sent immediately over the network from P_A to P_B. That data may then be held in the memory for P_B until transition B is ready to process it. There is no run-time overhead incurred in finding an available processor. Thus we see that static assignment provides the advantage of eliminating the run-time overhead to find a processor to execute a given transition as well as greatly reducing times for communication and waiting time for data.

A disadvantage of static assignment is that the load may not be well balanced among the processors in the network. It may happen that some of the processors are idle waiting for data while others are overworked, thus creating bottlenecks.

In our scheme of quasi-static assignment, an initial static assignment is automatically generated. Performance is monitored during execution. If throughput or latency requirements are not being met, a new static assignment is chosen. The process of changing from one static assignment to another we call *reassignment*. During reassignment we accomplish the change of each transition's assignment from its old assigned processor to its new assigned processor while execution of other transitions not being assigned to new processors continues with little interruption.

The static assignments are determined automatically, thus relieving the graph-writer of any concern about the architecture of the processor network. This not only increases human productivity in application development; it also reduces the cost of porting applications from one processor network to another. The method used to determine the static assignments is not included in this paper.

We will describe how we accomplished reassignment. A major challenge in doing this was the strict maintenance of the sequence of data elements in each place.

We begin with a discussion of the enhancements in PGM over the dataflow model. We continue with a description of inter-processor communication to support execution in the context of static assignment. Finally we discuss reassignment as an extension of static assignment.

3. PGM Enhancements over Dataflow

Among the several PGM enhancements over dataflow, we discuss a few that are germane to this paper. In doing so, we introduce some terminology.

A. In PGM, we combine some of the elements of Petri Nets [2] with those of data flow [1]. Specifically, in PGM, we refer to *transitions* and *places* instead of the *nodes* and *directed arcs* of data flow. Moreover, PGM defines two kinds of places, called *queues* and *graph variables*. The queue of PGM is exactly the same as the directed arc of data flow, in which data tokens are stored first-in/first-out. A graph variable stores a single token. When a transition produces a token to a graph variable, the previous token is destroyed, and the new token takes its place. A transition reads only the currently stored token in a graph variable. Let A be a transition and Q a place. If A produces data to Q, then we say that A is *upstream* from Q and that Q is *downstream* from A. Likewise, if A reads data from Q, then we say that A is *downstream* from Q and that Q is *upstream* from A.

B. During execution, a transition reads a number of tokens from each of its upstream places and performs some computations. Based on the results of the computations, the transition then produces a number of tokens to each of its downstream places. Finally, the transition consumes a number of tokens from each if its upstream places. In each upstream place, the number of tokens consumed may differ from the number of tokens that were read.

   In a queue the tokens consumed are destroyed, and the remaining tokens continue to be stored first-in/first/out. In a graph variable, consuming tokens has no effect.

C. In dataflow, each queue may contain a finite but unlimited number of data elements, or *tokens*. In PGM, we assign a *capacity* to each queue, which is the maximum number of tokens that may be stored in the queue. The PGMT system sets and maintains the queue capacity automatically during run-time as a means of controlling the scheduling of transitions for execution.

   We define *content* of a queue to be the number of tokens currently stored in the queue.

   We define *available capacity* to be (capacity – content). To be ready for execution (in addition to requiring sufficient content in every upstream queue), PGMT requires that in every downstream queue the number of anticipated output tokens not exceed the available capacity.

   A graph variable never inhibits the execution of an upstream or downstream transition. Thus, we define the content, capacity, and available capacity of a graph variable to be infinite. If a transition reads multiple tokens from a transition during a single execution, then multiple copies of the currently stored token will be supplied to the transition. If a transition produces a string of multiple tokens to a graph variable during a single execution, then only the last token will be stored in the graph variable.

D. In dataflow, each queue connects from a unique upstream node to unique downstream node. In PGMT terminology, this means that each queue has exactly one upstream transition and exactly one downstream transition. In PGM we relax this constraint and allow a given place (queue or graph variable) to have zero or more upstream transitions and zero or more downstream transitions. Henceforth in this document we refer to a place with exactly one upstream transition and exactly one downstream transition as a *simple place*.

   If a queue has zero downstream transitions, then there is no way that tokens can be removed from the queue. Thus if the queue reaches its capacity, then its available capacity is zero, and no upstream transition that would produce one or more tokens will be able to execute.

   If a queue has two or more downstream transitions, any one downstream transition may execute, provided the queue has sufficiently many tokens. After execution of that downstream transition, the queue's content may be reduced, thus causing other previously ready downstream transitions to become not ready.

   If a queue has zero upstream transitions, then there is no way that tokens can be produced to the queue. Thus, if the queue becomes empty, then no downstream transition that must read one or more tokens from the queue will be able to execute.

   If a queue has two or more upstream transitions, any one upstream transition may execute, provided the queue has sufficient available capacity (i.e., by executing and producing output tokens, that upstream transition will not cause the queue to exceed its capacity). After execution of that upstream transition, the queue may, as a result, have insufficient available capacity for any other upstream transition to execute, thus causing the other previously ready upstream transitions to become not ready.

In the next few sections we describe the management of data for a PGM graph in which all places are simple queues, each queue having a fixed capacity. After covering this case, we will discuss modifications to cover graph variables and non-simple places.

4. Processing in a Single Processor

The management of transition execution is based on the occurrence of events. Before we discuss distributed processing, we discuss the sequence of events that occurs during the execution of a transition.

In PGMT there is a module called the PEP (PGM Execution Program), which performs a number of different services within the PGMT system. For execution within a single processor, the PEP serves the following function:

- The PEP keeps track of the transitions that are ready to execute. Each transition is responsible to keep track of its own readiness to execute and to notify the PEP whenever its readiness changes. When the PEP decides to execute a transition, the PEP selects one that is ready, and causes it to execute. The PEP controls the capacity of every queue as a way of determining which transitions are ready to execute.

We now discuss the sequence of events that occur during the execution of a transition. In this discussion, we assume that there are three transitions A, B, and C and queues Q1 and Q2, connected as shown in Figure 1.

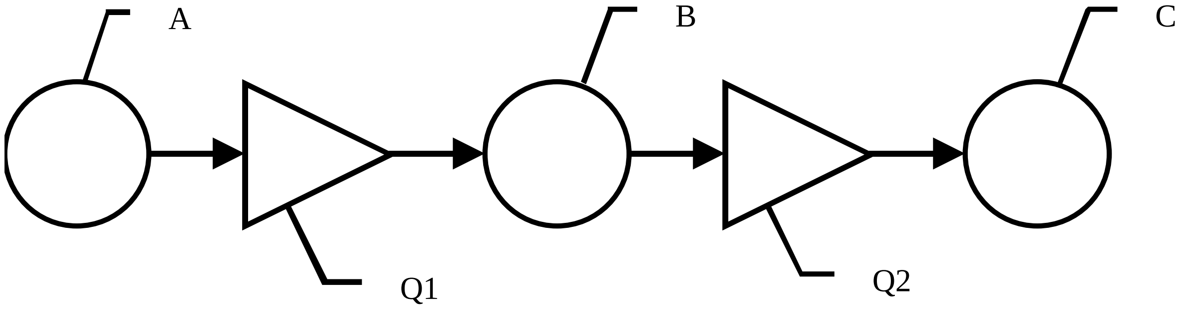

Figure 1

Assume that Q1 has sufficient content and Q2 has sufficient available capacity. When transition B executes, the following sequence of events occurs:

1. Transition B reads the tokens it needs for execution from Q1. This is a non-destructive read, and the tokens remain in Q1.
2. Transition B performs its processing, during which time it creates the output tokens to be produced to Q2.
3. Transition B produces its respective output tokens to Q2. Q2 stores the tokens and updates its content and available capacity.
   a. Q2 then notifies its upstream transition B of the change in available capacity. In response, B notes the change but defers notification of the PEP until the end of its current execution.

   b. Q2 also notifies its downstream transition C of the change in content. In response, C determines whether its readiness to execute has changed (i.e., from not ready to ready). If so, then C notifies the PEP of this change.
4. Transition B notifies its upstream queue Q1 to *consume* (i.e., to delete) a specified number of its tokens. Q1 deletes the specified tokens and updates its content and available capacity.
   a. Q1 notifies its upstream transition A of the change in available capacity. In response, A determines whether its readiness has changed. If so, it notifies the PEP.
   b. Q1 notifies its downstream transition B of the change in content. In response, B notes the change but defers notification of the PEP until the end of its current execution.
5. Transition B determines whether it is ready for a subsequent execution. If so, it notifies the PEP. NOTE: When the PEP initiated execution of B, it cleared B from being ready, and so if B (after execution) is ready for a subsequent execution, it must so notify the PEP.
6. Transition B notifies the PEP that its execution is complete and passes some parameters about its execution that the PEP uses to monitor performance.

After the execution of transition B is complete, the PEP performs some other functions and then selects a transition (possibly B, if it is again ready) for execution.

It may happen that the PEP changes the capacity of a queue. When this happens, the queue's available capacity changes, and so the queue notifies its upstream transition of the change in available capacity. When so notified the transition evaluates its readiness and notifies the PEP if its readiness has changed.

4.1. Modification for Graph Variables

In the PGMT design, the identification of a place as either a queue or graph variable is hidden from the transition. Thus the place must respond to read, produce, and consume requests from a transition in the proper manner, depending on whether it is a queue or graph variable. As noted above, a read of multiple tokens from a graph variable results in reading a string of multiple copies of the currently stored token. Producing a string of one or more tokens to a graph variable causes the destruction of the previously stored token and storage of the last token in the string. Consuming some number of tokens in a graph variable has no effect.

When a token is produced to a graph variable, this may affect the processing of its downstream transition. Indeed, it may even affect its readiness to execute. For example, the value of the new token may affect the number of tokens to be read or consumed from one of the transition's upstream queues. Step 3 above must be modified so that when a new token is produced, if the place is a queue, then the stated actions occur. Else, if the place is a graph variable, the graph variable notifies its downstream transition of the change. The downstream transition responds as described.

The consumption of tokens in a graph variable does not affect its available capacity. Thus we redefine Step 4 accordingly to indicate that the stated actions of Q1 in response to the consume request occur only if the place is a queue.

We will give a complete revision of the above sequence of events following the discussion about non-simple places.

4.2. Modification for Non-Simple Places

As described above, a non-simple place is one in which there are zero or more upstream transitions and zero or more downstream transitions. Modification of the above sequence of events simply means that a place must notify all of its upstream and downstream transitions whenever an event occurs that requires such notification.

Now assume that Q1 and Q2 are non-simple places (i.e., each is either a queue or graph variable, with possibly additional upstream and downstream transitions that are not shown in Figure 1). The following sequence of events occurs when transition B executes:

1. Transition B reads the tokens it needs for execution from Q1. This is a non-destructive read, and the tokens remain in Q1.
2. Transition B performs its processing, during which time it creates the output tokens to be produced to Q2.
3. Transition B produces its respective output tokens to Q2. If Q2 is a queue, then Q2 stores the tokens and updates its content and available capacity.
   a. Q2 then notifies all its upstream transitions (B and others) of the change in available capacity. In response, B notes the change but defers notification of the PEP until the end of its current execution. Each upstream transition B' other than B determines whether its readiness to execute has changed (i.e., from ready to not ready). If so, then B' notifies the PEP of this change.
   b. Q2 notifies each of its downstream transitions C of the change in content. In response, each downstream transition C determines whether its readiness to execute has changed (i.e., from not ready to ready). If so, then C notifies the PEP of this change.

   If Q2 is a graph variable and at least one token was produced, then Q2 destroys the previously stored token and stores the last token produced.
   a. Q2 does not notify its upstream transitions, because there is no change in the available capacity.
   b. Q2 notifies each of its downstream transitions C. In response, each downstream transition C determines whether its readiness to execute has changed (i.e., from not ready to ready or from ready to not ready). If so, then C notifies the PEP of this change.
4. Transition B notifies its upstream queue Q1 to *consume* (i.e., to delete) a specified number of its tokens. If Q1 is a queue, then Q1 deletes the specified tokens and updates its content and available capacity. As a result,
   a. Q1 notifies all its upstream transitions A of the change in available capacity. In response, A determines whether its readiness has changed. If so, it notifies the PEP.
   b. Q1 notifies all its downstream transitions (B and others) of the change in content. In response, B notes the change but defers notification of the PEP until the end of its current execution. Each downstream transition B' other than B determines whether its readiness to execute has changed (i.e., from ready to not ready). If so, then B' notifies the PEP of this change.

   If Q1 is a graph variable, then no change in the token stored occurs, and no notification of other transitions occurs.
5. Transition B determines whether it is ready for a subsequent execution. If so, it notifies the PEP. NOTE: When the PEP initiated execution of B, it cleared B from being ready, and so if B (after execution) is ready for a subsequent execution, it must so notify the PEP.

6. Transition B notifies the PEP that its execution is complete and passes some parameters about its execution that the PEP uses to monitor performance.

5. Distributed Processing with Static Assignment

Some processor networks have shared memory, meaning that there is a single memory bank to which all processors have access. In other processor networks the memory is not shared, each processor having its own dedicated memory. In the following discussion, we assume that the memory is not shared. Without confusion, we will say that data is *stored in a processor* to mean that the data is stored in the memory that is dedicated to that processor.

In PGMT every transition and every place of the application graph has an instance in every processor. By this we mean that the code for each transition and for each place is resident in every processor. In some processors, the code for a given transition or place may or may not execute, depending on the static assignment.

In a given static assignment, each transition and each place is assigned to a unique processor. Each place is assigned to the same processor as its downstream transition. This means that the tokens in the place are stored in the same processor where the downstream transition will execute. If the place's upstream transition is assigned to a different processor, then the place's code in that processor is called an *agent*.

For example, let Q be a place with upstream transition A and downstream transition B. Then $P_Q = P_B$ and either $P_A = P_Q$ or $P_A \neq P_Q$. The instance of Q in P_Q is called the *assigned* Q. If $P_A \neq P_Q$, then the instance of Q in P_A is called the *agent* of Q. If $P_A = P_Q$, then Q has no agent. For every processor P other than P_A and P_B, we say that the instance of Q in P is *idle*. During static assignment, the code for the idle instances of Q is not executed, but remains available in case of reassignment.

Note that if a place Q has multiple downstream transitions, then all downstream transitions of Q must be assigned to the same processor. If Q has multiple upstream transitions, then those upstream transitions may be assigned to different processors. In this case each processor with an upstream transition of Q also has an agent of Q.

The PEP code resides in every processor. The PEP code in each processor is called the *LocalPEP*. The LocalPEP has same responsibility given in the previous section, namely to keep track of the ready transitions, to select a ready transition, and initiate its execution.

There is a part of the PEP, called the *UberPEP*, which does the following:

- The UberPEP automatically determines the initial assignment of transitions to processors.

The UberPEP performs additional services related to reassignment. We will discuss these in the next section.

The UberPEP is separate from each of the LocalPEPs. The UberPEP communicates with each of the LocalPEPs by means of messages between processors. In this discussion we are not concerned with the details of communication between the UberPEP and the LocalPEPs.

Before graph execution starts, the UberPEP decides the initial assignment of every transition and notifies every LocalPEP about the initial assignment. Each LocalPEP, in turn, notifies the graph, which then notifies each node in the graph. At this time, each transition identifies whether it is assigned or idle. Each place identifies whether it is assigned, an agent, or idle, according to the rules given above, namely that each place is assigned to the same processor as its downstream transition(s). The assigned place also identifies the processor ID of each of its agents, and every agent identifies the processor ID of its assigned place.

In the case of distributed processing with static assignment, the sequence of events described in the previous section is the same, except for some modifications of steps 3 and 4. The differences lie entirely within the place. The underlying principle is that during execution of a transition, every inter-process communication occurs between an assigned place and its agent. Thus, to coordinate events in different processors, it is necessary and sufficient for a message to be sent either from a place's agent to the assigned place or from the assigned place to the agents. This is very important to understand and to remember.

No agent stores any tokens. However the agent does keep track of its content, capacity, and available capacity. These quantities are kept by the assigned place, which sends messages to the agents so that the agents may update these quantities.

In the above sequence of events in a transition execution, steps 3 and 4 must be modified to coordinate between the assigned place and its agents. We refer to the same graph segment in Figure 1, noting that the three transitions A, B, and C, may be assigned to different processors. Unless otherwise stated, the actions in this sequence occur in the processor P_B, to which transition B is assigned:

1. Transition B reads the tokens it needs for execution from Q1. This is a non-destructive read, and the tokens remain in the place. [Note that Q1 and B are assigned to the same process. Thus there is no change in this step.]
2. Transition B performs its processing, during which time it creates the output tokens to be produced to Q2.
3. Transition B produces its respective output tokens to Q2.
   If Q2 is assigned to P_B, then Q2 stores the tokens and updates its content and available capacity.
   If Q2 is a queue, then
   a. Q2 then notifies all of its locally assigned upstream transitions (i.e., those upstream transitions assigned to P_B) of the change in available capacity. In response, B notes the change but does not notify the LocalPEP of its readiness at this time. Each locally assigned upstream transition B' other than B evaluates its readiness to execute. If the readiness of B' changes, then B' notifies the LocalPEP of this change.
   b. Q2 sends a message to each of its agents notifying them of the change in content. Each agent then updates its content and available capacity and notifies each locally assigned upstream transition B" of the change in available capacity. In response, each locally

assigned upstream transition B' other than B evaluates its readiness to execute. If the readiness of B" changes, then B" notifies its LocalPEP of this change.

c. Q2 notifies each of its downstream transitions C of the change in content. In response, C determines whether its readiness to execute has changed (i.e., from not ready to ready). If so, then C notifies the LocalPEP of this change.

If Q2 is a graph variable, then the change in the value of a graph variable's stored token cannot affect the readiness of an upstream transition. Thus no notification of locally assigned transitions occurs, and no messages are sent to agents. We perform only Step b above in the case where Q2 is a queue.

Otherwise Q2 is an agent in P_B and is assigned in P_C. The agent of Q2 updates its content and available capacity.

If Q2 is a queue, then

a. The agent of Q2 notifies each of its locally assigned upstream transitions of the change in available capacity. In response, B notes the change but defers notification to the LocalPEP until transition execution is complete. Each locally assigned upstream transition B' evaluates its readiness to execute. If its readiness changes, then B' so notifies the LocalPEP.

b. The agent of Q2 sends the tokens in a message to the assigned Q2 in P_C.

c. Upon receiving this message, the assigned Q2 stores the tokens and updates its content and available capacity.

d. The assigned Q2 then notifies each downstream transition C of the change in content. In response, C determines whether its readiness to execute has changed (i.e., from not ready to ready). If so, then C notifies the LocalPEP of this change.

e. The assigned Q2 notifies each locally assigned upstream transition B" of the change in available capacity. In response, B" evaluates its readiness to execute. If its readiness has changed, it so notifies the LocalPEP.

f. The assigned Q2 sends a message to each of its agents other than the one assigned P_B (i.e., the one that sent the tokens), notifying the agent of the change in content. Each agent receiving this message updates its content and capacity and notifies its locally assigned upstream transitions. In response, each transition evaluates its readiness to execute and notifies the LocalPEP if there is a change.

If Q2 is a graph variable, then as before, there is no need to notify upstream transitions of the change. Thus, we perform only Steps b, c, and d described above in the case where Q2 is a queue.

4. Transition B notifies its upstream queue Q1 to *consume* (i.e., to delete) a specified number of its tokens. Q1 deletes the specified tokens and updates its content and available capacity.

If $P_A = P_B$, then Q1 has no agent. In this case,

a. Q1 notifies its upstream transition A of the change in available capacity. In response, A determines whether its readiness has changed. If so, it notifies the LocalPEP.

b. Q1 notifies its downstream transition B of the change in content. In response, B notes the change but does not notify the LocalPEP of its readiness at this time.

Otherwise $P_A \neq P_B$, and Q1 in P_A is an agent. In this case,

a. The assigned Q1 in P_B sends a message to its agent in P_A with the number of tokens consumed.

b. Upon receiving this message, the agent of Q1 updates its content and available capacity.

   c. The agent of Q1 then notifies its upstream transition A of the change in available capacity.
   d. The assigned Q1 in P_B notifies its downstream transition B of the change in content. In response, B notes the change but not notify the LocalPEP of its readiness at this time.
5. Transition B determines whether it is ready for a subsequent execution. If so, it notifies the LocalPEP. NOTE: When the LocalPEP initiated execution of B, it cleared B from being ready, and so if B (after execution) is ready for a subsequent execution, it must so notify the LocalPEP.
6. Transition B notifies the LocalPEP that its execution is complete and passes some parameters about its execution that the LocalPEP uses to monitor performance.

The code for each place, being resident in every process, is conditional, depending on whether the instance of the queue is assigned, is an agent, or is idle in the local process. All of the code is present in every process. Table 1 gives the place's response to each event. We abbreviate *graph variable* by *gvar*.

This completes the discussion of graph execution in a distributed system with static assignment.

Event	Response in local process for a queue	Response for a gvar
Upstream transition produces tokens.	If local queue is idle, then error. If local queue is agent, then • Update content and available capacity. • Notify local upstream transitions of new available capacity. • Send the tokens to the assigned queue. If local queue is assigned, then • Store the tokens. • Update content and available capacity. • Notify locally assigned upstream transitions of new available capacity. • Send message to each agent with change in content. • Notify downstream transitions of new content.	If local gvar is idle, then error. If local gvar is agent, then • Send the last token to the assigned gvar. If local gvar is assigned, then • Store the last token. • Notify downstream transitions of the change.
Downstream transition consumes tokens.	If local queue is idle, then error. If local queue is agent, then error. If local queue is assigned, then • Delete the tokens • Update content and available capacity. • Notify each locally assigned upstream transition of new available capacity. • Send a message to every agent with the change in content. • Notify each downstream transition of new content.	If local gvar is idle, then error. If local gvar is agent, then error. If local gvar is assigned, then • Do nothing.
Message from agent arrives with tokens.	If local queue is idle, then error. If local queue is agent, then error. If local queue is assigned, then • Store the tokens. • Update content and available capacity. • Notify locally assigned upstream transitions of new available capacity. • Send message to each agent with change in content (no message to the agent that sent the message with tokens). • Notify downstream transitions of new content.	If local gvar is idle, then error. If local gvar is agent, then error. If local gvar is assigned, then • Store the token (message should have only one token). • Notify downstream transitions of the change.
Message from assigned place arrives with change in content	If local queue is idle, then error. If local queue is assigned, then error. If local queue is agent, then • Update content and available capacity. • Notify upstream transition of new available capacity.	Error.

Table 1

6. Reassignment

In this section we address the implementation of reassignment. As in the case of execution with static assignment, the bulk of the work to accomplish reassignment is done by means of communication between the instances of each place in the different processors. First we will describe the sequence of events in a specific example. Then we will discuss the various cases that can occur, each case requiring its own variation of the event sequence. Finally, as we did above for static assignment we will give a table of the various events and the response to each.

In general terms, graph execution occurs with static assignment until it is determined that performance is not satisfactory and a different static assignment will do better. At that point we continue graph execution while dynamically reassigning the nodes to their newly assigned processors. The intent is to continue processing with minimal disruption to the execution. After the reassignment has been completed, we continue with the new static assignment until yet another reassignment is deemed necessary. We refer to this as *quasi-static assignment*.

In addition to the previously described function of determining initial assignment, the UeberPEP does the following:

- The LocalPEP collects performance information collected from the transitions that execute in its processor and transmits this information to the UeberPEP. The UeberPEP analyzes this information and decides whether a new assignment of transitions to processors is needed to improve performance.
- If a new assignment is needed, the UeberPEP determines a new assigned processor for each transition and initiates the reassignment by sending the new assignment to every LocalPEP. Each LocalPEP then notifies the local graph of this reassignment.
- Each local graph notifies the transitions and places in its processor, which then implement the reassignment. When the reassignment is complete, the local graph notifies the LocalPEP that reassignment is complete.
- Upon notification that reassignment is complete in each processor, the LocalPEP sends a message to the UeberPEP. The UeberPEP is constrained not to initiate another reassignment until after all LocalPEPs have reported that the previous reassignment is complete.

Reassignment begins in each processor when the LocalPEP notifies the graph to initiate reassignment, giving a list of the transitions together with the processor to which each transition is reassigned. Note that this list includes every transition, even if it is reassigned to its old assigned processor.

In response the graph sets a counter to the total number of nodes (i.e., transitions and places) in the graph. The graph notifies all of the transitions of their reassigned processors and then notifies all of the places. These notifications trigger a sequence of events in each node that continues as graph processing continues. When each node completes its reassignment in the local processor, it so notifies the graph, causing the graph to decrement its counter. When the counter becomes zero, reassignment is complete in the local processor, and the graph notifies the LocalPEP.

Upon notification to start reassignment, each transition in the local processor does the following:

- If it was assigned to the local processor,
    - if it will be assigned to this processor, then remain enabled for execution.
    - if it will be assigned to a different processor, then disable execution.
- If it was not assigned to this processor,
    - if it will be assigned to this processor, then enable execution.
    - if it will be assigned to a different processor, then remain disabled.
- In all cases, report immediately to the graph that the transition has completed its reassignment.

Before we discuss the general case of places that may be either queues or graph variables, either simple or non-simple, we give a brief description of the case with a simple queue. Figure 2 depicts an example of reassignment of a queue Q with upstream transition A and downstream transition B.

Upon notification to start reassignment, each place instance in a given processor has nine possible combinations of roles before and after reassignment. The place instance may initially be assigned, an agent, or idle, and may have any of the same three roles after reassignment.

In the this example, if N is a node and P is a processor, we write N(P) to denote the instance of the node N in processor P. Thus if A is a transition, then A(P3) denotes the instance of A in processor P3. Assume that processors P1, P2, P3, and P4 are distinct and that two transitions A and B are connected by a queue Q as illustrated in the following drawing, where A is reassigned from P1 to P3, and B is reassigned from P2 to P4:

The dotted arrows show messages between the instances of Q during static assignment: between Q(P1) and Q(P2) before reassignment and between Q(P3) and Q(P4) after reassignment. In particular, before reassignment, the messages from Q(P1) to Q(P2) contain tokens produced by A(P1), and the messages from Q(P2) to Q(P1) contain messages that tell when tokens are consumed during the execution of A(P2). After reassignment, the same communications will occur between Q(P3) and Q(P4).

There are several message types that are designed to synchronize the activities of the instances of Q in each processor. Each such message is sent from the instance of Q in one processor to the instance of Q in another processor.

For each place, the events of reassignment are independent of the other places. The reassignment events for all places occur concurrently and asynchronously. Except as noted, transitions continue to execute during reassignment. The LocalPEP continues all of its normal activities for graph execution while awaiting notification that reassignment is complete.

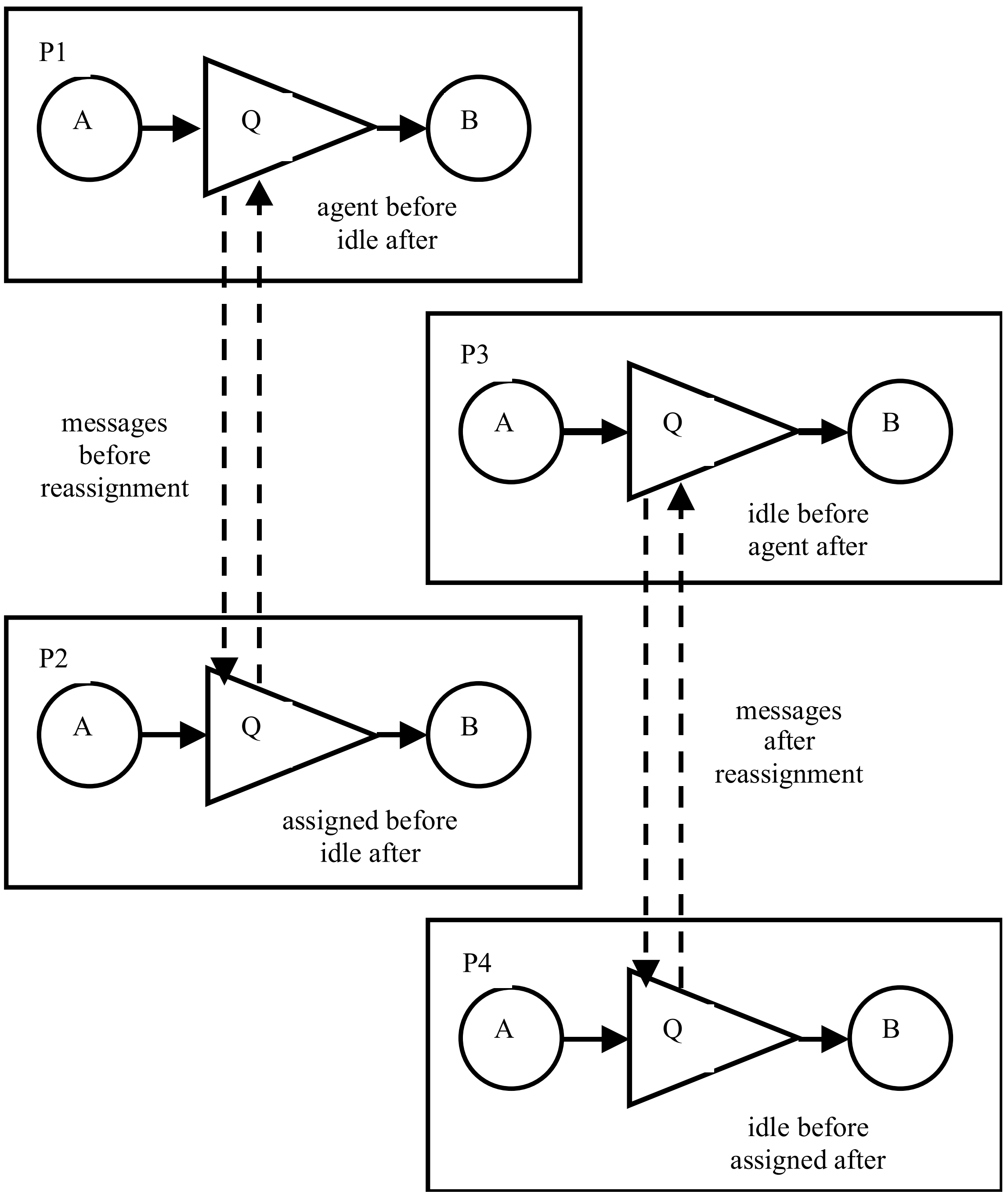

Figure 2

Table 2 gives the sequences of events that occur in each of the four processors.

By default we set the content and available capacity of all idle places to zero. Thus, during reassignment, a newly assigned transition cannot be made ready until notified that there is sufficient content upstream and available capacity downstream.

P1 (Q was an agent, becomes idle)	P2 (Q was assigned, becomes idle)	P3 (Q was idle, becomes an agent)	P4 (Q was idle, becomes assigned)
Send "NO MORE TOKENS" message to the old assigned Q in P2 to indicate that Q in P1 will send no more token messages.	Send a message "NO MORE CONSUMES" to the old agent of Q in P1 to indicate that no more consume messages will be sent.	Wait for "ASSIGNED READY" message from Q in P4.	As token messages arrive from P2, store the tokens, update the content and available capacity, and notify the local downstream transition of the new content.
Wait for "NO MORE CONSUMES" message from the old assigned Q in P2.	Forward all currently stored tokens to the new assigned Q in P4. Include the current capacity.	Update the content and available capacity.	Wait for "ASSIGNED DONE" message from Q in P2.
Notify the local graph that Q has completed reassignment.	As tokens arrive from the old agent in P1, forward them to the new assigned Q in P4.	Notify the upstream transition A in P3 of the current capacity.	Send "ASSIGNED READY" message to the new agent in P3. Include the content and capacity.
	Wait for "NO MORE TOKENS" message from the old agent in P1.	Notify the local graph that Q has completed reassignment.	Notify the local graph that Q has completed reassignment.
	Send "ASSIGNED DONE" message to the new assigned Q in P4 to indicate that no more tokens will be forwarded.		
	Notify the local graph that Q has completed reassignment.		

Table 2

Note that the messages being sent between the various instances of Q to coordinate the reassignment. These messages ensure that all tokens in Q(P2) (i.e., originally produced by A(P1) to Q(P1) and sent from Q(P1) to Q(P2)) are forwarded from Q(P2) to Q(P4) before transition A(P3) is allowed to start executing. This sequence also ensures that all expected events for reassignment occur in each processor PN before Q(PN) notifies the graph in PN that it has completed reassignment.

This example illustrates the sequences of events in each of four processors to accomplish reassignment in one of several cases. Other cases are variations on the theme.

For example, it may happen that the upstream transition A is not reassigned (i.e., A is reassigned from P1 to P1, so that the agent of Q remains in the same processor) while the downstream transition B is reassigned from P2 to P4. In this case, Q(P1) was an agent and remains an agent, while Q(P2) was assigned and becomes idle and Q(P4) was idle and becomes assigned. The agent Q(P1) must stop sending tokens to Q(P2) and start sending them to Q(P4), but only after the appropriate messages have been received indicating that no more of the normal message for the old static assignment will be sent. We call these *done messages*.

For another example, it may happen that A is reassigned from P1 to P2 while B is reassigned from P2 to P1. Then Q(P1) was the agent and becomes assigned, while Q(P2) was assigned and becomes the agent.

To implement reassignment, all possible cases must be covered. The problem is further complicated by the fact that events in a given processor may not occur in the prescribed order.

In the above table, suppose that reassignment is initiated in processor P1 long before initiation in P2. This could cause Q(P2) to receive the "tokens done" message from Q(P1) before the initiation of reassignment in P2. This is a race condition that the design of reassignment must account for. There are many similar race conditions that can occur during the course of reassignment in the processor network.

During static assignment, each instance of a place in a processor plays one of three roles: assigned, agent, or idle. To be prepared for reassignment, each instance must also know which processor contains its assigned instance and which processors contain its agents. When reassignment is initiated, each place instance identifies the processors containing its new assigned instance and its new agent. In doing this, the queue identifies its new role and the sequence of events for it to assume its new role. All of the code for reassignment of queues must be resident in every processor.

Further complications exist to account for the more general cases of a place possibly being a graph variable or a non-simple place. This discussion is designed to account for the general case.

We use the model of a finite state machine to design reassignment for places. Figure 3 illustrates the various states and the events that trigger movement from one state to another. This diagram captures the essential nature of what must be done to accomplish the reassignment of a queue.

The labeled rectangles represent states of the place during reassignment. The solid arrows represent movements from one state to another. The dotted arrows represent messages that are sent from the instance of the place in one processor to the place in another processor. These messages are the ones that trigger transition from one state to the next during reassignment.

The states "ASSIGNED", "AGENT", and "IDLE" represent the various roles that the place plays during normal execution with static assignment. There are four additional states (indicated by "AWAITING … ") that are used to indicate that a message must arrive in order to move to the next state. When the message has arrived, the place takes certain actions and moves to the next state, depending on its ultimate state.

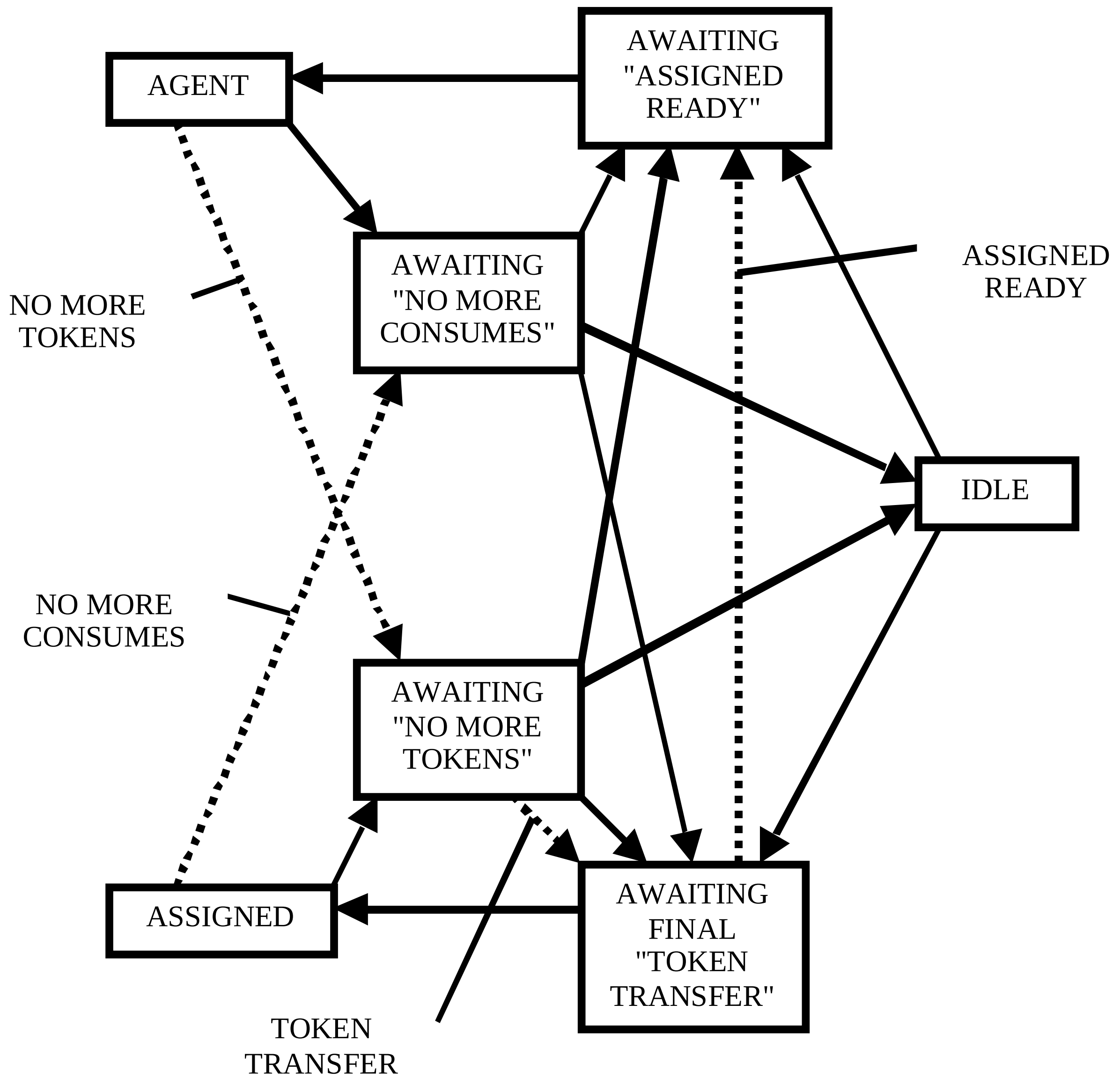
AGENT
AWAITING "ASSIGNED READY"
AWAITING "NO MORE CONSUMES"
ASSIGNED READY
NO MORE TOKENS
IDLE
NO MORE CONSUMES
AWAITING "NO MORE TOKENS"
ASSIGNED
AWAITING FINAL "TOKEN TRANSFER"
TOKEN TRANSFER

Figure 3

Upon notification of reassignment, each instance of the place in each processor identifies its current state and its destination state. The combination identifies a unique path of intermediate states from its initial state to its final state. In each intermediate state the expected message must arrive before the place can move to the next state.

To implement this finite state machine model, recognizing that messages may arrive at any time (possibly before they are expected), we define four event flags, corresponding to the awaited messages. By default, each event flag is cleared. When a message arrives, we process the message, set the respective event flag, and then call a procedure called *REASSIGN*.

The REASSIGN procedure contains a loop. In the first pass through the loop, this procedure identifies the initial and final roles and checks for the special cases. If appropriate, it sends a message, moves the place to the next state, and repeats the loop. On the next pass through the loop, it checks the respective flag. If the flag not set, then the REASSIGN procedure terminates. If the flag is set, then (if appropriate) the REASSIGN procedure sends another message and moves the place to the next state. On the final pass through the loop, the REASSIGN procedure moves the place to the final role, clears all the flags, reports to the local graph that reassignment is complete and puts the place in the initial state awaiting notification of the next reassignment.

Before we give the detailed algorithm for the REASSIGN procedure, we discuss the contents and the rationale of the four different messages indicated in the above diagram. There is an event flag associated with each message. This flag is clear by default, and it is set to indicate that the message has been received (or set within the REASSIGN procedure if no such message is expected). We use the adjectives *old* and *new* to mean "before reassignment" and "after reassignment" respectively.

NO MORE TOKENS: This is a simple message with no additional content. It is sent from each of the place's old agents to the old assigned place upon initial notification of reassignment. It indicates that execution of each upstream transition in its old assigned processor has been suspended and that no more messages with tokens will be sent. Each old assigned place that is AWAITING "NO MORE TOKENS" must wait for such a message from every agent. Thus we implement this event flag as a counter. We initially set this counter to the number of agents. When a NO MORE TOKENS message arrives from an old agent, we decrement the counter. When the counter becomes zero, the old assigned place may move to the next state.

NO MORE CONSUMES: This is a simple message with no additional content. It is sent from the old assigned place to each of its old agents upon initial notification of reassignment. It indicates that execution of the downstream transitions in their common old assigned processor has been suspended and that no more tokens will be consumed – hence no more messages to the old agent indicating the consumption of tokens.

TOKEN TRANSFER: This passes tokens from the old assigned place to the new assigned place so that these tokens may be read and used by the downstream transitions in the newly assigned processor. Note that there may be several such messages if the old assigned queue continues to receive token messages from its old agents. This message indicates the following:

1. That execution of the old assigned downstream transitions have terminated.
2. That the new downstream transition may begin execution.

FINAL TOKEN TRANSFER: The old assigned place sends this message only after receiving NO MORE TOKENS from every agent. It passes on any remaining tokens together with the place capacity.

ASSIGNED READY: This is sent from the new assigned place its new agents. Its purposes are

1. To indicate that the new assigned place is ready to accept tokens from the new agents.
2. To pass on the current place capacity and place content to each new agent.

When one of these messages arrives, the place processes the information contained in the message, sets the respective flag or decrements the counter, as appropriate, and calls its REASSIGN procedure.

In the following algorithm for the REASSIGN procedure, we use the adjectives *initially* and *finally* to indicate the state of the place's local instance before and after reassignment. For example, we say that it is *initially ASSIGNED* to mean that it was assigned to the local processor before reassignment; we say that it is *finally AGENT* to mean that it is an agent after reassignment.

```
Repeat forever {
    If in state INITIAL {
        Identify processors where upstream and downstream transitions are assigned before and after
            reassignment. (Note: This information is used to identify the intermediate states as well as
            any special cases to be addressed.)
        Identify initial and final roles (ASSIGNED, AGENT, IDLE).

        If initially ASSIGNED {
            If (finally ASSIGNED)  and
                (there is at most one upstream transition whose old and new assigned processors are the
                same) {
                Move to state FINAL.  /* nothing to do */
            }
            Else{
                If finally ASSIGNED or finally AGENT {
                    Block old local upstream transitions
                        (i.e., temporarily prevent them from executing.  We do this by notifying the
                    transition that the available capacity is 0.)
                }
                Send message NO MORE CONSUMES to all old agents.
                Move to state AWAITING "NO MORE TOKENS".
            }
        }  /* end If initially ASSIGNED  */

        If initially AGENT{
            If finally AGENT and
                (no more than one upstream transition) and
                (old assigned and new assigned are in the same processor) {
                Move to state FINAL.  /* nothing to do */
            }
            Else {
                Block local upstream transitions.
                Send NO MORE TOKENS message old assigned.
                Move to state AWAITING "NO MORE CONSUMES".
            }
        }  /* end If initially AGENT  */

        If initially IDLE {
            If finally ASSIGNED {
                Move to state AWAITING "TOKEN TRANSFER".
            }
        If finally AGENT {
            Move to state AWAITING "ASSIGNED READY".
        }
        If finally IDLE {
            Move to state FINAL.  /* nothing to do */
        }
    }  /* end of initially IDLE  */
}  /* end If  in state INITIAL

If in state AWAITING "NO MORE TOKENS" {  /* must be old assigned with existing old agent */
    If NO MORE TOKENS counter ≠ 0 {
        Return.  /* Terminate the routine – to be called again upon arrival of message. */
    }
    Else {  /* NO MORE TOKENS counter = 0, proceed to the next state */
        If finally ASSIGNED {
            Send ASSIGNED READY to new agents.
            If initially ASSIGNED {
                Set flag TOKEN TRANSFER.
            }
            Move to state AWAITING "TOKEN TRANSFER".
        }
        Send TOKEN TRANSFER to new assigned
        If finally AGENT {
            Move to state AWAITING "ASSIGNED READY".
        }
        If finally IDLE {
            Move to state FINAL.
        }
    }
}  /* end If in state AWAITING "NO MORE TOKENS"  */

If in state AWAITING "NO MORE CONSUMES" {  /* must be old agent */
    If Flag NO MORE CONSUMES not set {
        Return.  /* Terminate the routine – to be called again upon arrival of message. */
    }
    Else {  /* Flag NO MORE CONSUMES is set, proceed to the next state */
        If finally ASSIGNED {
            Move to state AWAITING "TOKEN TRANSFER".
        }
        If finally AGENT {
            Move to state AWAITING "ASSIGNED READY".
        }
        If finally IDLE {
            Move to state FINAL.
        }
    }
    } /* end If in state AWAITING "NO MORE CONSUMES" */

    If in state AWAITING "FINAL TOKEN TRANSFER" {  /* must be new assigned */
        If flag FINAL TOKEN TRANSFER not set {
            Return.  /* Terminate the routine – to be called again upon arrival of message. */
        }
        Else {  /* Flag TOKEN TRANSFER is set, proceed to the next state */
            If initially ASSIGNED or initially AGENT {
                Unblock local upstream transitions.
            }
            Move to state FINAL.
        }
    }  /* end If in state AWAITING "TOKEN TRANSFER" */

    If in state AWAITING "ASSIGNED READY" {  /* must be new agent */
        If Flag ASSIGNED READY not set {
            Return.  /* Terminate the routine – to be called again upon arrival of message. */
        }
        Else {  /* Flag ASSIGNED READY is set, proceed to the next state */
            If initially AGENT or initially ASSIGNED {
                Unblock local upstream transition.
            }
            Move to state FINAL.
        }
    } /* end If in state AWAITING "ASSIGNED READY" */

    If in state FINAL {
        /* all tasks are completed for reassignment of place in local processor */
        /* clear flags and clean up for normal execution */
        If finally IDLE {
            Set content and capacity to 0.
        }
        Clear all event flags.
        Set NO MORE CONSUMES counter to number of new agents.
        Move to state INITIAL.
        Report to graph that reassignment is complete.
        Return.
    }  /* end If in state FINAL */
}  /* end Repeat forever  */
```

Note that a message may be sent from a place instance that has not been notified of reassignment and received at a place for which reassignment is ongoing. Another possibility is that a message for reassignment may be sent from a place instance that has been notified and is received in a place instance that has not yet been notified of the reassignment. The processing of all messages must account not only for these possibilities but also for any contingency that may arise.

Table 3 shows responses to messages for a given place that would be sent from a place instance during normal execution in an environment of static assignment. It may arrive at the respective place instance after reassignment has begun. We show the processing in response to each such message. To keep it simple, we describe the response for a queue. The response for a graph variable is to be adjusted as described above.

Incoming Message	Rationale	Processing in Response when receiving this message
Tokens	From agent to assigned place – message contains tokens produced by upstream transition assigned to other processor.	If in reassignment If awaiting NO MORE TOKENS and is old assigned and is not new assigned, then forward incoming tokens to new assigned. Else error. Else (not in reassignment) If not assigned, throw error. Store the token Notify local upstream transitions Notify downstream transitions Send capacity update message to agents other than one sending the message.
Change content	From assigned place to agent to update the content and available capacity when downstream transition consumes tokens.	If not a queue, throw error. If in reassignment Ignore the message – do nothing. Else (not in reassignment) If not an agent, throw error. Update content and available capacity. Notify local upstream transitions.

Table 3

Table 4 shows responses to messages for a given place that would be sent from a place instance during reassignment. Such a message may arrive at the respective place instance before reassignment has begun. We show the processing in response to each such message.

Incoming Message	Rationale	Processing in Response when receiving this message
NO MORE TOKENS	From old agent to old assigned place. Indicates that no more message with tokens will come from the sending agent.	Decrement the "NO MORE TOKENS" counter. If not in reassignment and local place not assigned, Throw error. If in reassignment If not old assigned, throw error. Call REASSIGN.
NO MORE CONSUMES	From old assigned place to old agent. Indicates that no more content or capacity update messages will come.	Set the "NO MORE CONSUMES" flag. If not in reassignment and local place not agent, Throw error. If in reassignment If not old agent, throw error. Call REASSIGN.
TOKEN TRANSFER	From old assigned place to new assigned place. Passes on tokens for new assigned transition to process.	Store the incoming tokens. If in reassignment If not new assigned, throw error. Update content and available capacity. Notify local downstream transitions.
FINAL TOKEN TRANSFER	From old assigned place to new assigned place. Indicates that no more TOKEN TRANSFER messages will come.	Store incoming tokens, if any are contained in message. Set the "FINAL TOKEN TRANSFER" flag. Update the capacity and available capacity. If in reassignment If not new assigned, throw error. Notify local upstream and downstream transitions Send "ASSIGNED READY" message to new agents. Call REASSIGN.
ASSIGNED READY	From new assigned place to new agent. Indicates that new assigned place may begin receiving new tokens from new agents. Message includes current capacity and content.	Set the "ASSIGNED READY" flag. If in reassignment If not new agent, throw error. Update the content, capacity, and available capacity. Notify local upstream transitions. Call REASSIGN.

Table 4

Notes about the actual implementation of reassignment in PGMT:

The above discussion is intended to give the underlying logic of accomplishing reassignment. To make it understandable we tried to keep this discussion simple. In doing so, we glossed over some details. The actual implementation in PGMT somewhat more complex than the process described above:

1. We recognized that reassignment is disruptive to normal processing, because some transitions are blocked from execution. In order to reduce such disruption, we bypassed some of the intermediate states where it was possible to do so without endangering the integrity of the token sequence in each place. This necessarily made the algorithm more complicated.
2. The actual implementation was initially designed and implemented without a full appreciation of all the needed events to ensure a clean reassignment. Thus, during the testing phase we discovered problems with the design and made fixes "on the fly". As a result, the actual implementation of the REASSIGN procedure is more complicated than what is shown above.
3. The processors under the control of the Command Program do not have localPEPs. For purposes of this discussion, we refer to each such a processor as a *Command Program Processor.* Each processor with a localPEP is called a *Graph Processor*. It is important that during an assignment or reassignment, every Command Program Processor with an assigned graph input or output port participate in the reassignment. The reason is to ensure that data tokens be sent and/or received between these Command Program processors and the appropriate Graph Processors. For the initial assignment (i.e., before graph execution begins), this is not a problem, because the initial assignment occurs as part of graph construction. However, for reassignment, the localPEPs are disabled in the Command Program Processors and thus cannot notify the local graph about a reassignment. To solve this problem, we selected a dedicated Graph Processor called the *master processor*. When the local graph in the master processor is notified of a reassignment, that local graph sends a message to each of the local graphs in the Command Program Processors about the reassignment, including the new processor assignments of all the transitions. When each such Command Program Process completes reassignment, it sends a message back to the dedicated local graph in the master processor. In all other ways, reassignment occurs in these Command Program Processes in the same way as in a Graph Process. In addition to waiting for the conditions for completion of reassignment in its local processor, the master processor must wait until it has received completion messages from all the Command Program Processors.

References

1. *Processing Graph Method 2.1 Semantics*, by David J. Kaplan and Richard S. Stevens, July 29, 2002, hereinafter called the PGM Spec.
2. *Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing*, by Richard M. Karp and Raymond E. Miller, SIAM J. Appl. Math., Vo. 14, No. 6, November, 1966.
3. Peterson, J.L. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1981.